Building a Streaming Microservices Architecture

With Apache Spark Structured Streaming & Friends

Scott Haines
Senior Principal Software Engineer, Twilio

@newfront

A little about me.

- I work at Twilio building massive data systems
- I run a bi-weekly internal Spark Office Hours where I offer training and guidance to teams at the company
- >12 years working on large distributed analytics systems

@newfront

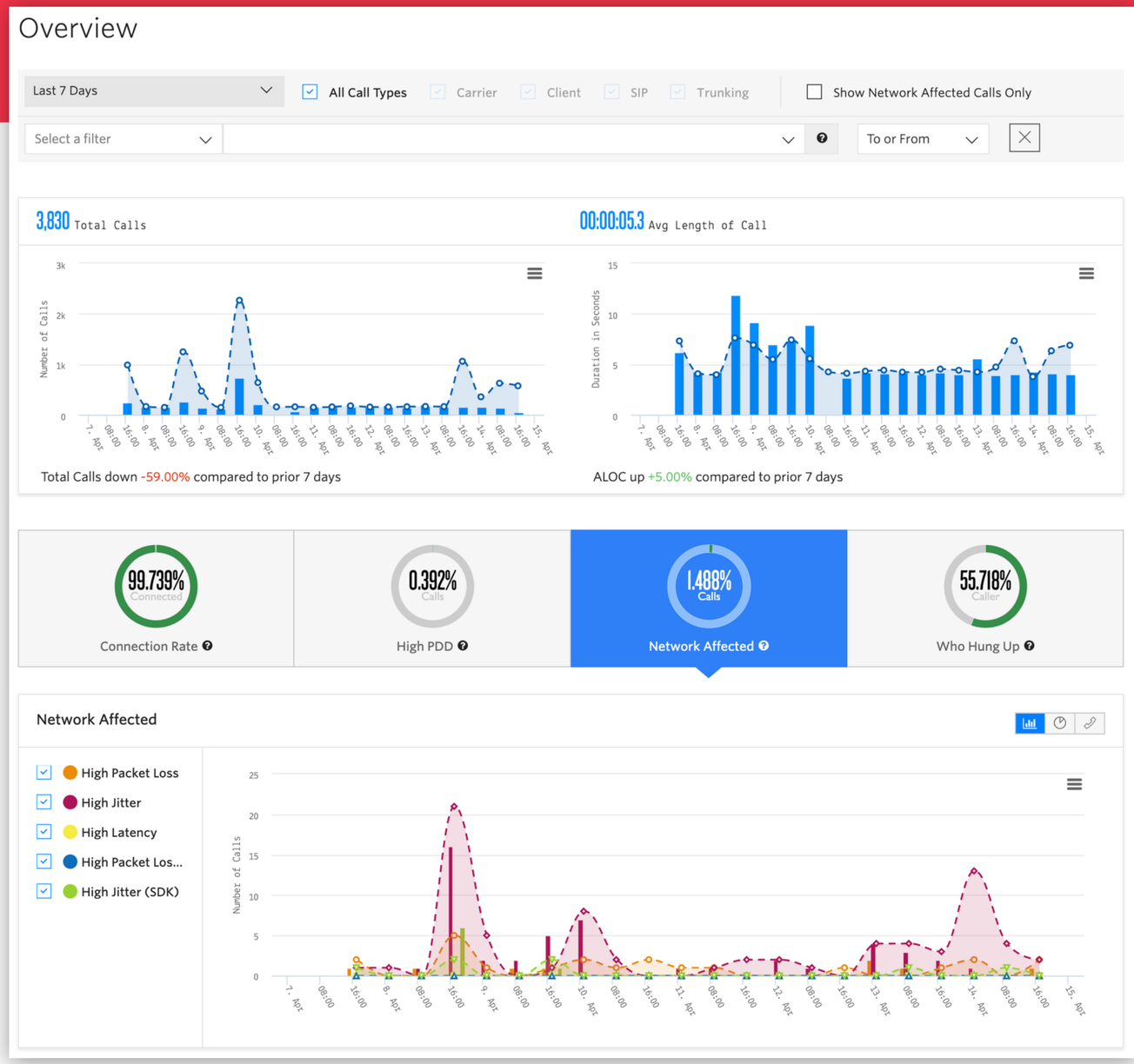
A little about me.

- I work at Twilio building massive data systems
- I run a bi-weekly internal Spark Office Hours where I offer training and guidance to teams at the company
- >12 years working on large distributed analytics systems
- Published work on Distributed Analytics Systems

@newfront

Voice Insights

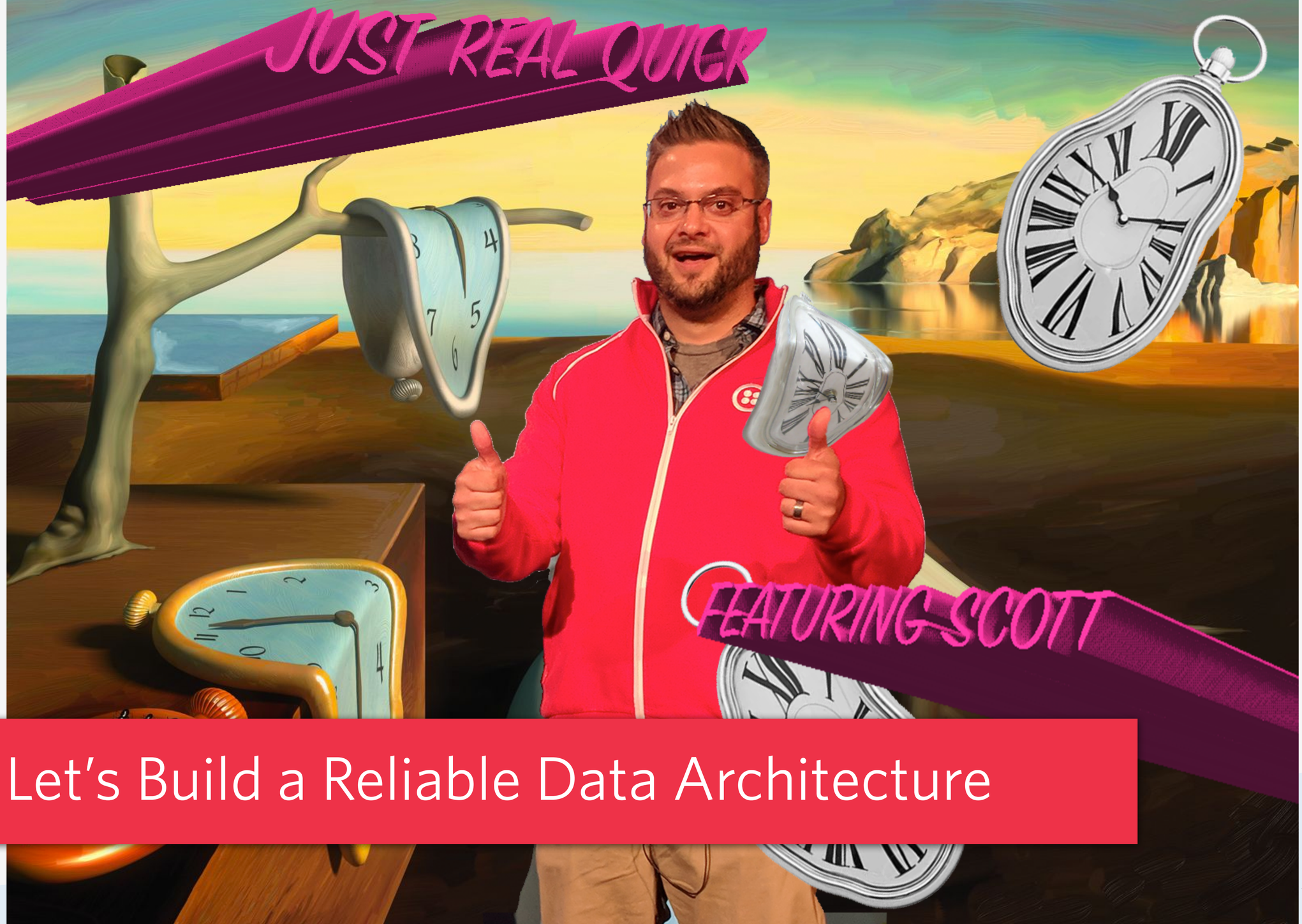
Accountable observability data and interactive analytics and insights for the voice business and customers.



twilio

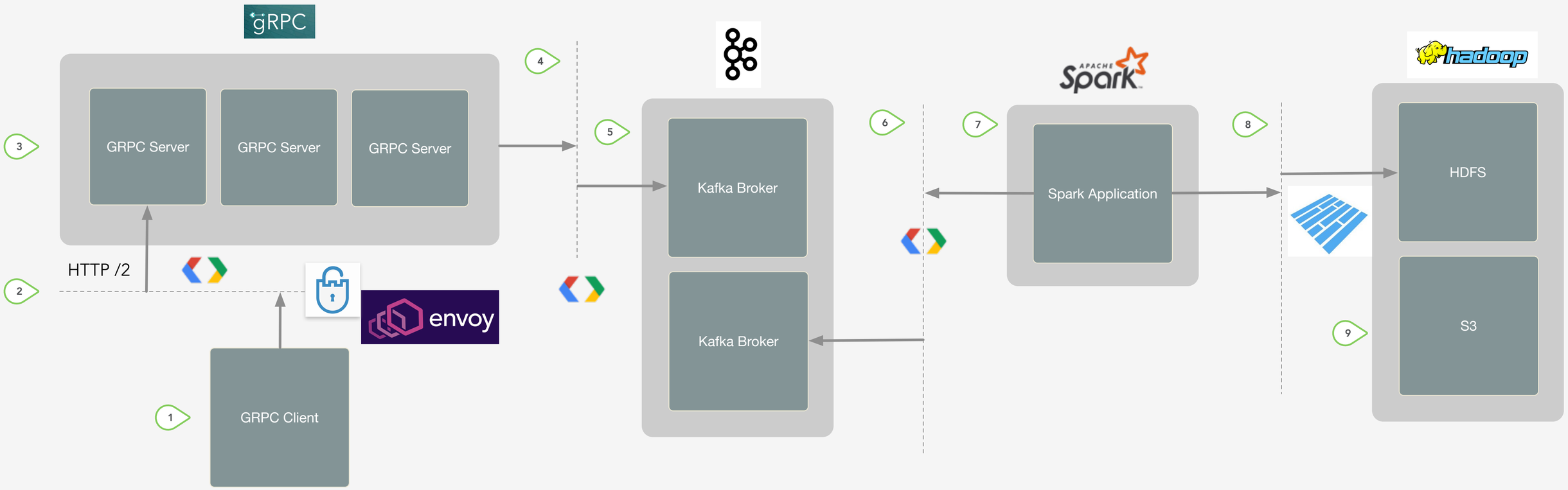
>1MIL

Events per Second



Let's Build a Reliable Data Architecture

Goal: Reliable E2E Streaming Data Pipeline



@newfront

Strong and Reliable Data **starts at Ingest**

Structured Data

- JSON has Structure.
- But JSON isn't **strictly** Structured Data.

>95%

Of people love JSON*.

```
{
  "type": "CallEvent",
  “call_sid”: "CA123",
  "attributes": [
    {
      “account_sid”: “AC123”
      “start_ms": 123,
      “end_ms": "435"
    }
  ]
}
```

@newfront @twilio

Structured Data

- JSON has **poor runtime guarantees** due to its flexible nature. **Optimize** for compile time guarantees.
- **Debugging** corrupt data in a large distributed system ruins hopes and dreams.

```
{
  "type": "CallEvent",
  “call_sid": “CA234”,
  “attributes":"oops"
}
```

100%

Of people saddened by bad data*

@newfront @twilio

Protocol Buffers

- Well Defined Events tell their own Story
- Type-Safety Rules
- **Versioning your API / Pipeline / Data** now just means sticking to a version of your schema
- Rely on **Releases for versioning**
- Interoperable with most major languages (java/scala/c++/go/obj-c/node-js/ python/...)

```
message CallEvent {
  uint64 created_ms    = 1;
  string call_sid      = 2;
  uint64 account_sid   = 3;
  EventType event_type = 4;
  Region region        = 5;
}
```

Protocol Buffers

Take Aways

- Data Accountability
- Lightning Fast Serialization / Deserialization
- **Plays with nicely gRPC**
- Interoperable with Spark SQL
- Like “JSON with Guard Rails”

```
message CallEvent {
  uint64 created_ms    = 1;
  string call_sid      = 2;
  uint64 account_sid   = 3;
  EventType event_type = 4;
  Region region        = 5;
}
```

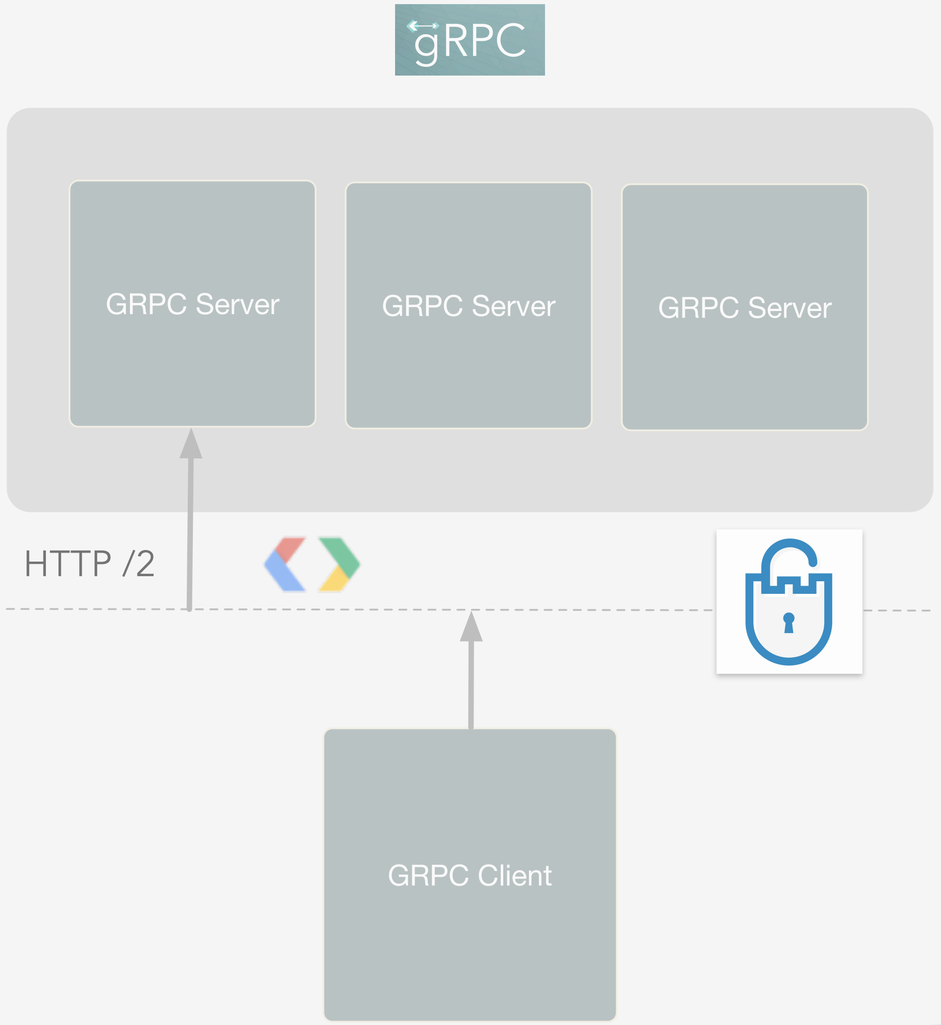
100%

Of people like when things work between releases!

Data Engineers love gRPC

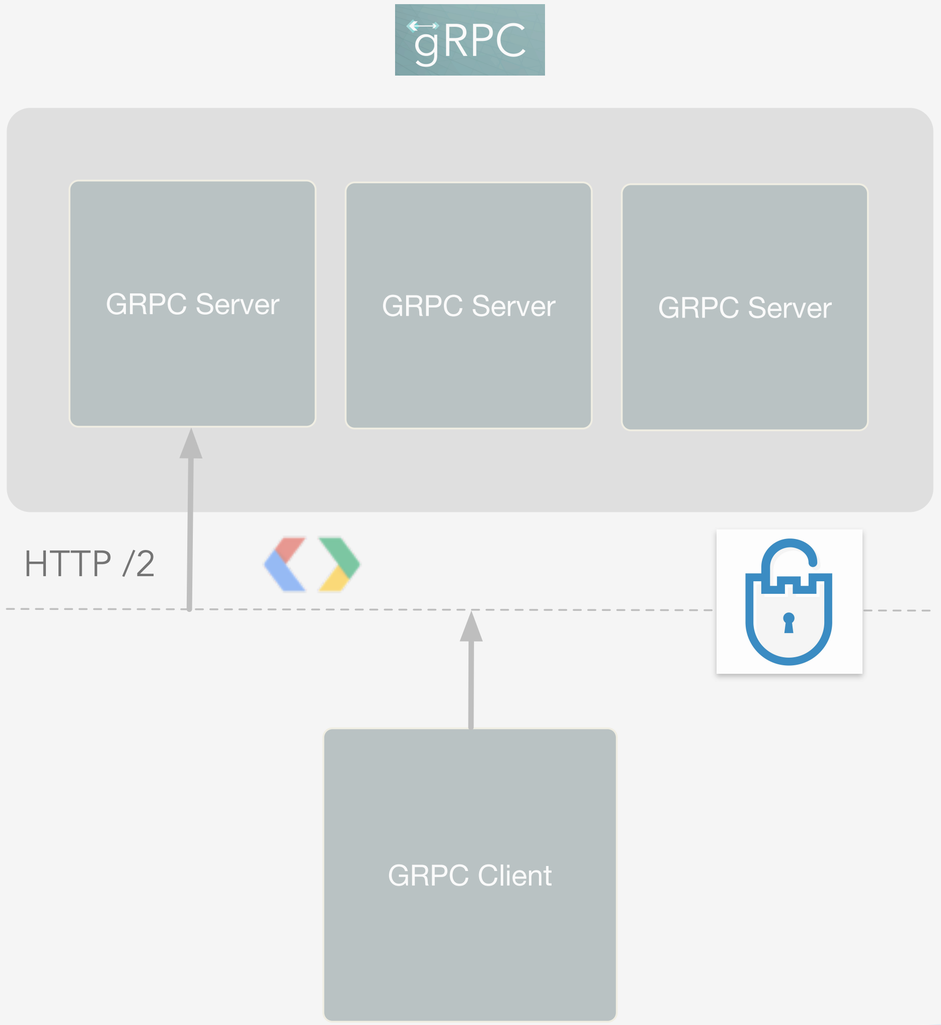
*technically not universally accepted

gRPC | saves time



gRPC | saves time

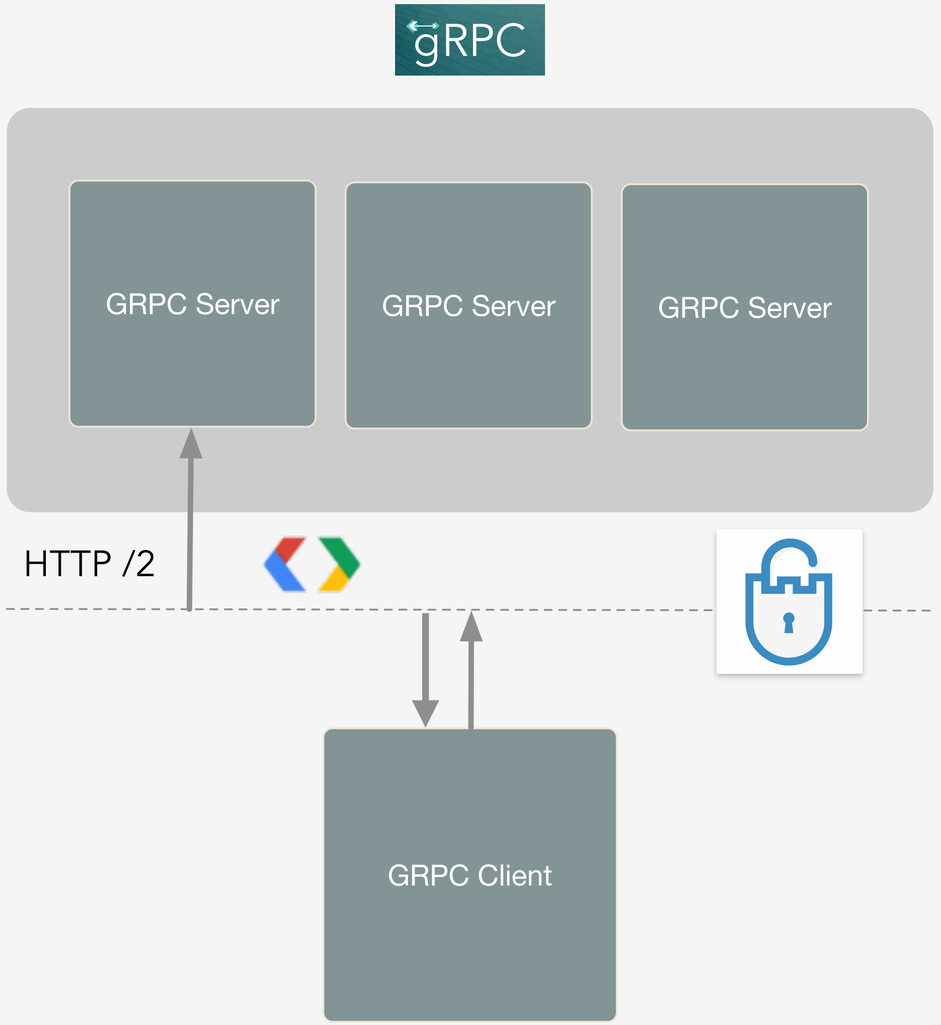
```
val time = “$$$”
```



GRPC.

GRPC // nutshell

- **RPC** = remote procedure call. “**G**” stands for generic or Google
- Great for Internal Services
- **High Performance**
  - Compact Binary Exchange Format
  - Compile Idiomatic API Definitions
- **Capable of Bi-Directional Streaming**
  - **Pluggable HTTP/2 transport**



GRPC.

- Building a CallEvent Service.
- 1: Define your messages (**call.proto**)

call.proto

```
1   syntax = "proto3";
2   package com.twilio.examples.dataai;
3
4   message CallEvent {
5       string call_sid      = 1;
6       string account_sid   = 2;
7       uint32 duration      = 3;
8       enum CallState {
9           unknown_state    = 0;
10          completed        = 1;
11          failed           = 2;
12      }
13      CallState call_state = 4;
14      enum Region {
15          unknown_region   = 0;
16          us1              = 1;
17          us2              = 2;
18          de1              = 3;
19      }
20      Region region        = 5;
21      uint64 event_time    = 6;
22  }
23
24  message CallSid {
25      string call_sid      = 1;
26  }
```

GRPC.

- Building a CallEvent Service.
- 1: Define your messages (call.proto)
- 2: Define your services (service.proto)

service.proto

```
1   syntax = "proto3";
2   package com.twilio.examples.dataai;
3
4   import "call.proto";
5
6   service TwilioEvents {
7       rpc trackCall(CallEvent) returns (CallSid);
8       rpc getCallInfo(CallSid) returns (CallEvent);
9   }
```

@newfront

GRPC.

- **Building a CallEvent Service.**
- 1: Define your messages (**call.proto**)
- 2: Define your services (**service.proto**)
- 3: Compile your messages and service stubs.


```
sbt clean compile package publishLocal
```

GRPC.

- Building a CallEvent Service.
- 1: Define your messages (call.proto)
- 2: Define your services (service.proto)
- 3: Compile your messages and service stubs.
- 4: Implement Traits and Run!

```
val ec: ExecutionContext = ExecutionContext.fromExecutorService(
  Executors.newFixedThreadPool( nThreads = 8)
)
val service = new TwilioEventService
val sb = ServerBuilder
  .forPort( port = 4000)
sb.addService(TwilioEventsGrpc.bindService(service, ec))

val server = sb.build()

sys.addShutdownHook {
  server.shutdownNow()
}
server.start()
server.awaitTermination()
```

GRPC.

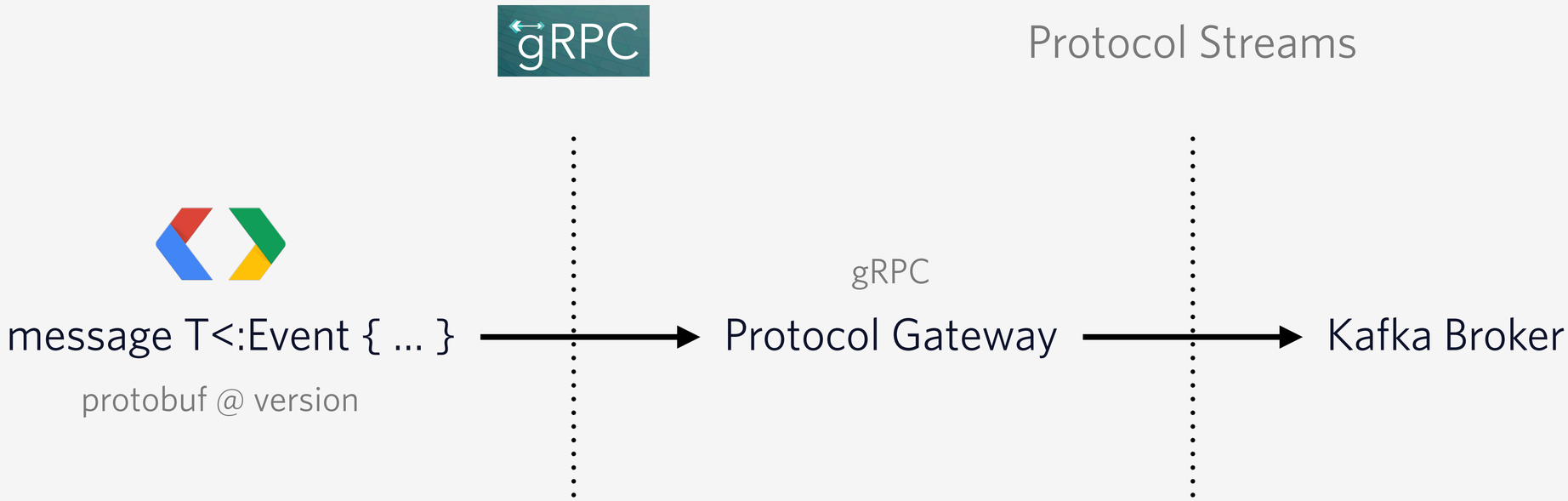
- Building a CallEvent Service.
- Client SDKs essentially write themselves.
- JSON <-> Protobuf is still possible for maintaining customer facing APIs

```
val ec: ExecutionContext = ExecutionContext.fromExecutorService(
  Executors.newFixedThreadPool( nThreads = 8)
)
val service = new TwilioEventService
```

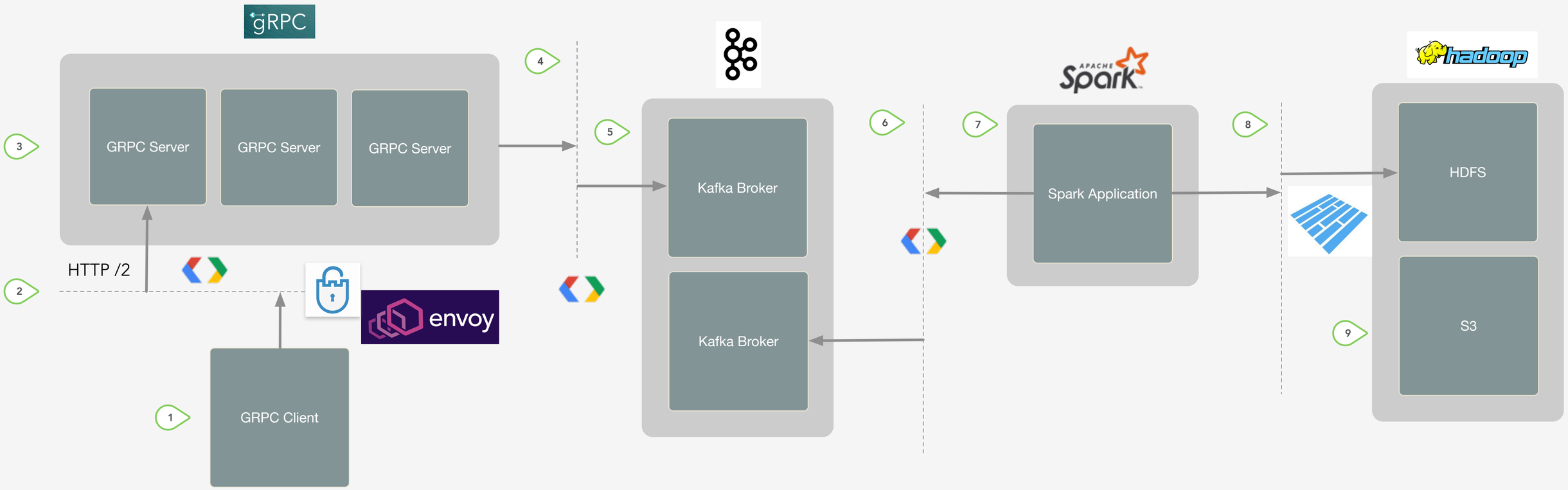
```
val channel = ManagedChannelBuilder
  .forAddress( name = "localhost",  port = 4000)
channel.usePlaintext()
val mc = channel.build()

val asyncStub = TwilioEventsGrpc.stub(mc)
val callEvent = CallEvent("CA123", "AC123", 300, CallState.completed, Regio
val send = asyncStub.trackCall(callEvent)
```

```
}
server.start()
server.awaitTermination()
```

Common Pattern Emerges

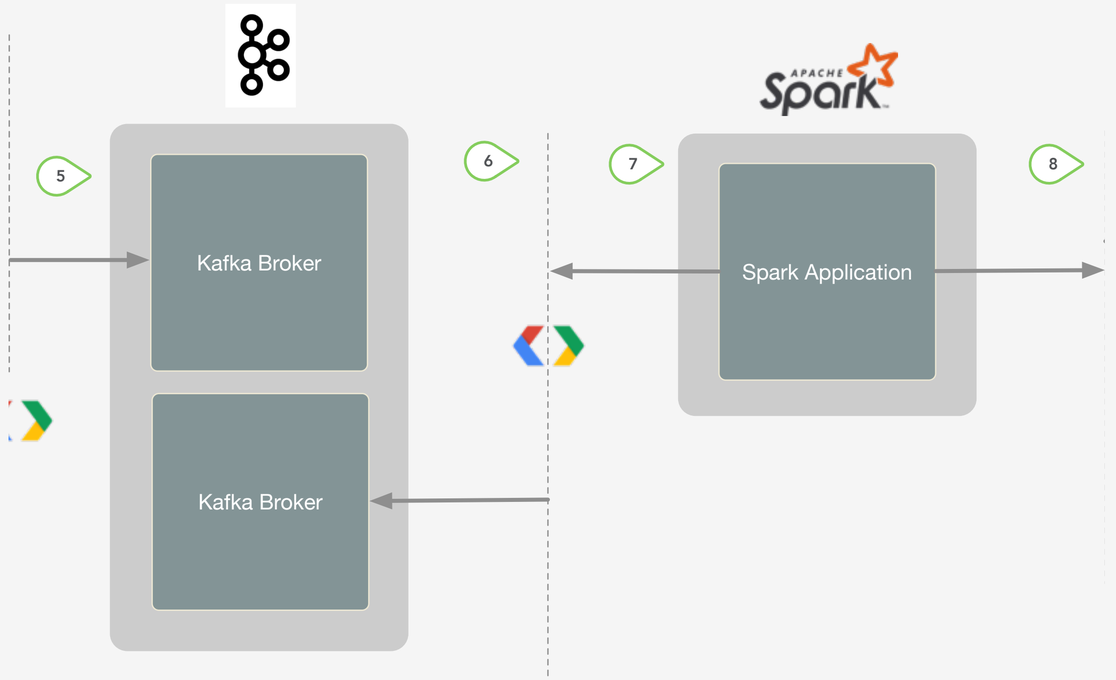


Still Building...



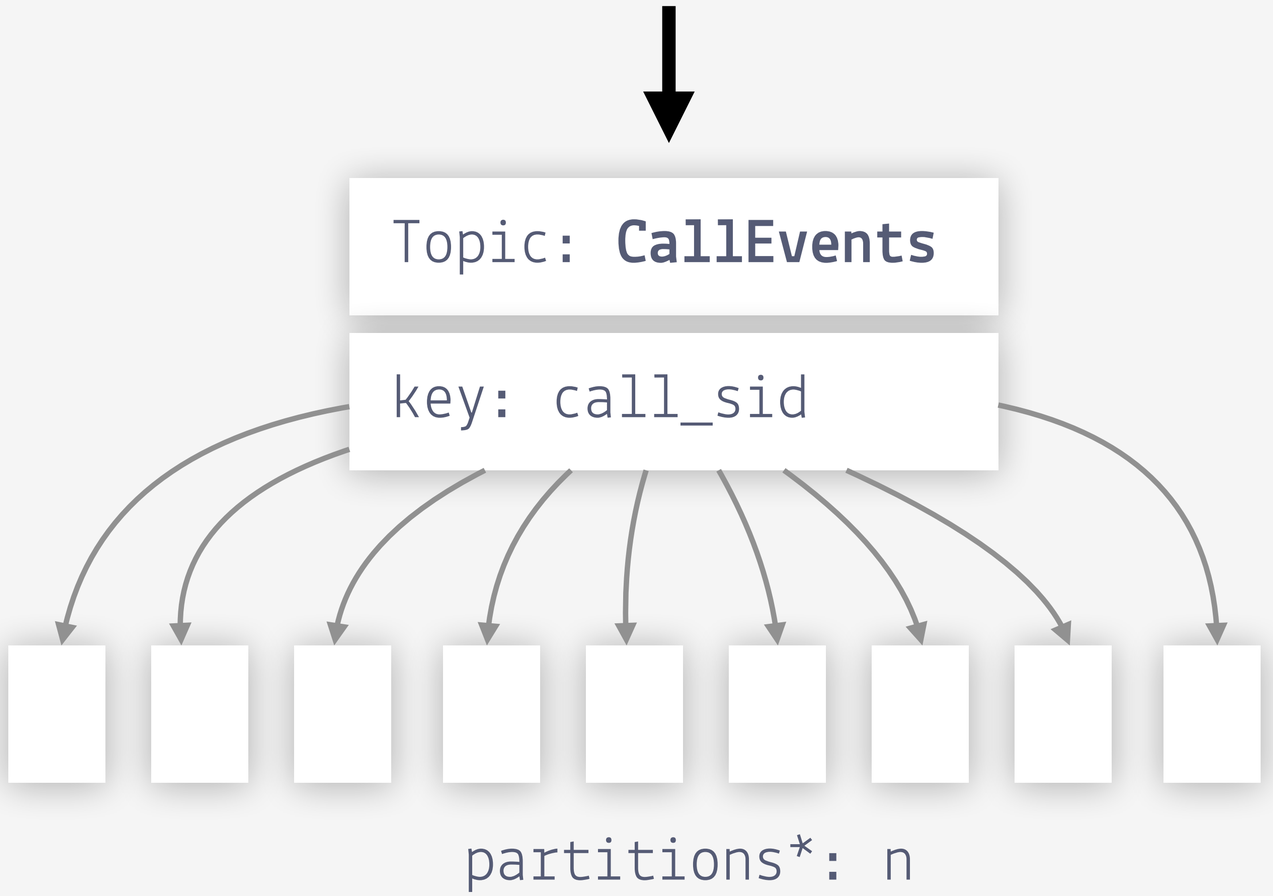
@newfront

Kafka + Protobuf + Spark



Kafka.

- Solve for common Data Pipeline Problems
- **Partition Keys** are important. Use them to your advantage
- **Spark AQE -** adaptive query execution can handle hot spots. Non-spark services not-so-much.



* number of partitions: factor of producer records/s * avg(record.bytesize)

Spark.

- Use ScalaPB's ExpressionEncoders to natively convert protobuf to Catalyst Optimizable DataFrames
- Marry this with Sparks Kafka DataSource Reader/Writer

```
import scalapb.spark.ProtoSQL
import ProtoSQL.implicits._
```

```
val spark = sparkSql
val startTime = Instant.now()

val publishData = Seq(
  CallEvent( "CA123", "AC123", 300, CallEvent.CallState.completed, CallEvent.Region.us1, startTime.t
  CallEvent( "CA234", "AC234", 500, CallEvent.CallState.completed, CallEvent.Region.us2, startTime.m
  CallEvent( "CA1234", "AC123", 130, CallEvent.CallState.completed, CallEvent.Region.de1, startTime.
)

ProtoSQL.createDataFrame(spark, publishData)                    : DataFrame
  .as[CallEvent]                                                : Dataset[CallEvent]
  .map(call2Kafka(_, topic))(kafkaDataEncoder)                  : Dataset[KafkaData]
  .write                                                        : DataFrameWriter[KafkaData]
  .format(source="kafka")                                       : DataFrameWriter[KafkaData]
  .option("kafka.bootstrap.servers", "192.168.64.2:32634")      : DataFrameWriter[KafkaData]
  .save()                                                       : Unit
```

```
(base) shaines@C02C71BVMD6V kafka % ./kafka-consumer.sh datasummit-click-stream
Kafka Broker List (k8s): 192.168.64.2:32634

CA234AC234? (0?????.

CA1234AC123? (0?????.

CA123AC123? (0?????.
```

Spark.

- Behind the Scenes...
- Spark is doing magic

```
ProtoSQL.createDataFrame(spark, publishData)
  .as[CallEvent]
  .map(call2Kafka(_, topic))(kafkaDataEncoder)
```

```
Encoders.product[KafkaData]
```

```
def call2Kafka(ce: CallEvent, topic: String): KafkaData = {
  KafkaData(ce.callSid.getBytes, ce.toByteArray, topic)
}
```

```
case class KafkaData(
  key: Array[Byte],
  value: Array[Byte],
  topic: String
)
```

```
== Physical Plan ==
* SerializeFromObject (4)
+- * MapElements (3)
   +- * DeserializeToObject (2)
      +- * LocalTableScan (1)


(1) LocalTableScan [codegen id : 1]
Output [6]: [call_sid#0, account_sid#1, duration#2, call_state#3, region#4, event_time#5L]
Arguments: [call_sid#0, account_sid#1, duration#2, call_state#3, region#4, event_time#5L]

(2) DeserializeToObject [codegen id : 1]
Input [6]: [call_sid#0, account_sid#1, duration#2, call_state#3, region#4, event_time#5L]
Arguments: com.twilio.examples.dataai.call.CallEvent$@42172065.messageReads.read.apply, obj#28: com.twilio.examples.dataai.call.CallEvent

(3) MapElements [codegen id : 1]
Input [1]: [obj#28]
Arguments: com.twilio.examples.dataai.SparkAppSpec$$Lambda$1355/0x0000000801613040@4ac77269, obj#29: com.twilio.examples.dataai.KafkaData

(4) SerializeFromObject [codegen id : 1]
Input [1]: [obj#29]
Arguments: [knownnotnull(assertnotnull(input[0, com.twilio.examples.dataai.KafkaData, true])).key AS key#30, knownnotnull(assertnotnull(input[0, com.twilio.examples.dataai.KafkaData, true])).value AS value#31, staticinvoke(c
```

Spark.

- End to End Tests ensure you can press the release button anywhere in the pipeline.
- Can be automated to ensure your Spark Apps can continue working with any changes to the upstream gRPC
- Can use for Canary testing updates

```
val callEventReader = spark.readStream
  .format( source = "kafka")
  .option("kafka.bootstrap.servers", "192.168.64.2:32634")
  .option("subscribe", topic)
  .option("kafka.maxOffsetsPerTrigger", "100")
  .option("startingOffsets", "earliest")

val callEventDFs = callEventReader.load()

val streamingQuery = callEventDFs                                                    : DataFrame
  .select( col = "value")                                                            : DataFrame
  .flatMap(row2callEvent)                                                            : Dataset[CallEvent]
  .withColumn( colName = "date", F.from_unixtime(F.col( colName = "event_time")/1000)) : DataFrame
  .writeStream                                                                       : DataStreamWriter[Row]
  .format( source = "memory")                                                        : DataStreamWriter[Row]
  .queryName( queryName = "calls")                                                   : DataStreamWriter[Row]
  .outputMode(OutputMode.Append())                                                   : DataStreamWriter[Row]
  .start()                                                                           : StreamingQuery


streamingQuery.processAllAvailable()

spark.sql( sqlText = "select * from calls").show()

streamingQuery.stop()
```

Spark.

- Use ScalaPB to read local streams
- Ensure your Spark Apps can continue working with any new protobuf dependencies
- Test simple reads and complex aggregations locally, deploy globally

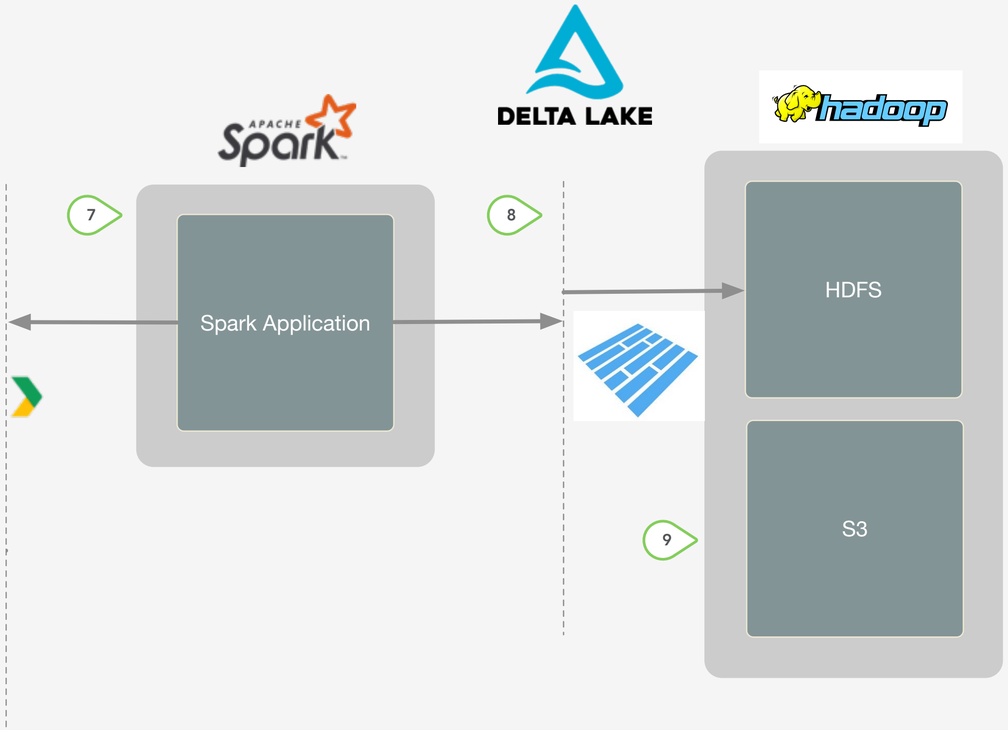
```
val callEventReader = spark.readStream
  .format( source = "kafka")
  .option("kafka.bootstrap.servers", "192.168.64.2:32634")
  .option("subscribe", topic)
  .option("kafka.maxOffsetsPerTrigger", "100")
  .option("startingOffsets", "earliest")

val callEventDFs = callEventReader.load()

val streamingQuery = callEventDFs                                    : DataFrame
  .select( col = "value")                                            : DataFrame
  .flatMap(row2callEvent)                                            : Dataset[CallEvent]
```

```
+--------+----------+--------+----------+------+------------+-------------------+
|call_sid|account_sid|duration|call_state|region|   event_time|               date|
+--------+----------+--------+----------+------+------------+-------------------+
|  CA1234|      AC123|     130| completed|   de1|1605671262014|2020-11-18 03:47:42|
|   CA234|      AC234|     500| completed|   us2|1605672222014|2020-11-18 04:03:42|
|   CA123|      AC123|     300| completed|   us1|1605673662014|2020-11-18 04:27:42|
+--------+----------+--------+----------+------+------------+-------------------+
```

```
spark.sql( sqlText = "select * from calls").show()

streamingQuery.stop()
```

@newfront

Spark & Beyond



Spark + Friends.

- Proto to Catalyst DataFrame
- Conversion to Parquet in Delta Table
- Partitioned by Date

```
val callEventReader = spark.readStream
  .format( source = "kafka")
  .option("kafka.bootstrap.servers", "192.168.64.2:32634")
  .option("subscribe", topic)
  .options(Map[String, String](
    elems = "kafka.request.timeout.ms" -> "60000",
    "kafka.session.timeout.ms" -> "120000",
    "kafka.fetch.max.wait.ms" -> "45000",
    "startingOffsets" -> "earliest",
    "failOnDataLoss" -> "false",
    "kafkaConsumer.pollTimeoutMs" -> "30000",
    "fetchOffset.numRetries" -> "20",
    "fetchOffset.retryIntervalMs" -> "100",
    "maxOffsetsPerTrigger" -> "6000000"
  ))

val callEventDFs = callEventReader.load()

val streamingQuery = callEventDFs                                                    : DataFrame
  .select( col = "value")                                                            : DataFrame
  .flatMap(row2callEvent)                                                            : Dataset[CallEvent]
  .withColumn( colName = "datetime", F.from_unixtime(F.col( colName = "event_time")/1000)) : DataFrame
  .withColumn( colName = "date", F.to_date(F.col( colName = "datetime")))            : DataFrame
  .writeStream                                                                       : DataStreamWriter[Row]
  .format( source = "delta")                                                         : DataStreamWriter[Row]
  .outputMode(OutputMode.Append())                                                   : DataStreamWriter[Row]
  .partitionBy( colNames = "date")                                                   : DataStreamWriter[Row]
  .option("checkpointLocation", checkpointLocation)                                  : DataStreamWriter[Row]
  .option("mergeSchema", "true")                                                     : DataStreamWriter[Row]
  .start(deltaTablePath)                                                             : StreamingQuery

streamingQuery.processAllAvailable()
streamingQuery.stop()
```

Spark + Friends.

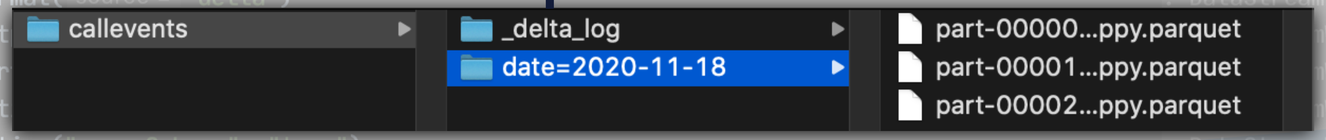
- Now it is easy for Downstream applications to pick up using Parquet Protocol Streams

```
val callEventReader = spark.readStream
  .format( source = "kafka")
  .option("kafka.bootstrap.servers", "192.168.64.2:32634")
  .option("subscribe", topic)
```

```
{
    "metaData": {
        "id": "b999af24-dc5e-48ba-b83e-6e76dd47d51c",
        "format": {
            "provider": "parquet",
            "options": {}
        },
        "schemaString": "{\"type\":\"struct\",\"fields\":[{\"name\":\"call_sid\",\"type\":\"string\",
        "partitionColumns": [
            "date"
        ],
        "configuration": {},
        "createdTime": 1605719040875
    }
}
```

```
val streamingQuery = callEventDFs
  .select( col = "value")
  .flatMap(row2callEvent)
  .withColumn( colName = "datetime", F.from_unixtime(F.col( colName = "event_time")/1000))
  .withColumn( colName = "date", F.to_date(F.col( colName = "datetime")))
  .writeStream
  .format( source = "delta")
  .option("mergeSchema", "true")
  .start(deltaTablePath)

streamingQuery.processAllAvailable()
streamingQuery.stop()
```

: DataFrame
: Dataset[CallEvent]
: DataFrame
: DataFrame
: DataStreamWriter[Row]
: StreamingQuery

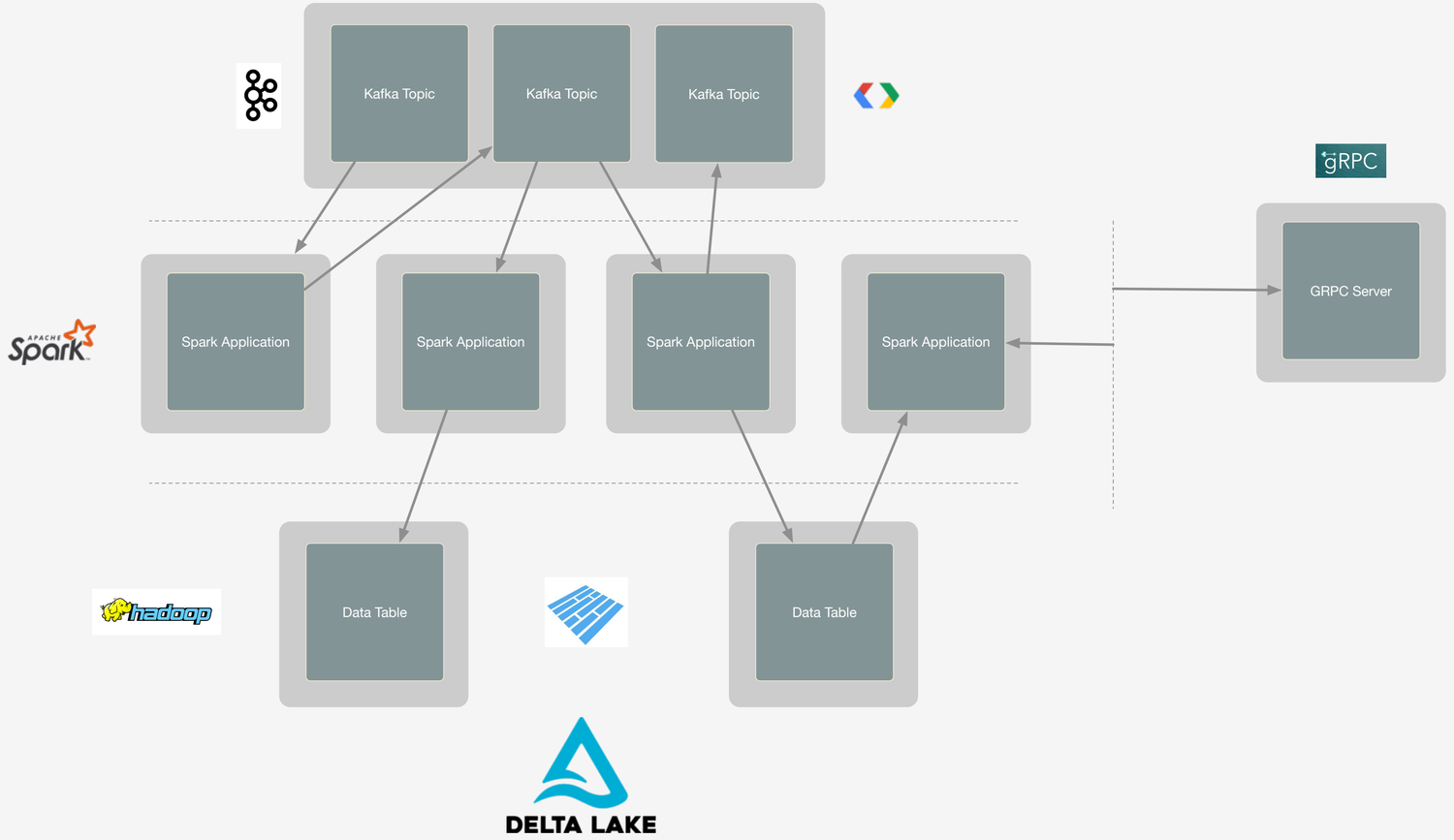


@newfront

Rinse and Repeat

Just keep adding new Flows.

1. Define your Structured Data
2. Define your service definitions
3. Emit Data / Enqueue to Kafka
4. Read and Drop in HDFS (delta) or pass along to a new Topic



THANK YOU

 @newfront